Name: Maura Kieft
ID: 103947905

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 6 **Spring 2019, CU-Boulder**

Quick links: 1a 1b 1c 2a 2b 2c 2d 3a 3b 3c

1. As a budding expert in algorithms, you decide that your semester service project will be to offer free technical interview prep sessions to your fellow students. Not surprisingly, you are immediately swamped with appointment requests at all different times from students applying many different companies, some with more rigorous interviews than others (i.e., some will need more help than others). Let A be the set of n appointment requests. Each appointment a_i in A is a pair $(start_i, end_i)$ of times and $end_i > start_i$. To manage all of these requests and to help the most student students that you can, you develop a greedy algorithm to help you manage which appointments you can keep and which ones you have to drop (you can only tutor one student at a time).

Name: Maura Kieft
ID: 103947905

(a) (2 points) Draw an example with at least 5 appointments where a greedy algorithm that selects the shortest appointment will fail.

If we had 5 appointments (1,4), (4,8), (3,5), (8,12), (7,9)

1 —— 4 4 —— 8 8 —— 12
3 —— 5 7 ——9

(3,5) and (7,9) are shorter intervals which would be chosen for the greedy algorithm that selects the shortest appointment although the optimal solution would be to chose the intervals (1,4), (4,8), and (8,12)

(b) (2 points) Draw an example with at least 5 appointments where a greedy algorithm that selects the longest appointment will fail.

If we had five appointments (1,3), (3,5), (5,7), (1,4), (4,7)

1 — 3 3 — 5 5 — 7
1 —— 4 4 —— 7

The greedy algorithm which selects the longest appointment would chose intervals (1,4) and (4,7) although the optimal solution would be intervals (1,3), (3,5), and (5,7).

(c) (6 points) Describe and prove correctness for a greedy algorithm that is guaranteed to choose the subset of appointments that will help the maximum number of students that you help.

Idea: Choose appointments in ascending order of end time. Initially, A is the set of all the appointment requests, and S is empty. The algorithm will sort in ascending order by end time. Since A has been sorted, then the first element will have the earliest end time. Add request i to S. Delete all the requests from A that aren't compatible with request i. When A i = n, return the set S as the set of accepted appointments.
Pseudocode:

```
appointmentSchedule(A, n)
    Sort(A, A+n, apptComp)        //sorts in ascending order of end time
                                  //(bool apptComp(a1.end < a2.end))
    S = empty
    prevEnd = -1                  //to check if next appt conflicts
    for(i = 1 to n)
       if(A[i].start > prevEnd)
          S.append(A[i])          //doesn't conflict, add to set
          prevEnd = A[i].end
    return S                      //return accepted appointments of set S
```

Theorem: This greedy algorithm provides an optimal solution in choosing the subset of appointments that will help the maximum number of students.
If S is an appointment schedule produced by our greedy algorithm, and S' is an optimal schedule, then for any $1 \leq i \leq |S|$ we will have f(i,S) $\leq$ f(i,S')
Proof by Induction: Base Case: The first appointment the algorithm selects has to be an appointment with end time no later than any appointment. So, f(1,S) $\leq$ f(1,S'). Assuming the claim holds for some i in $1 \leq i \leq |S|$, the ith appointment in S ends before the ith appointment in S' since f(i,S) $\leq$ f(i,S'). So, the (i+1)st appointment in S' must start after the ith appointment in S ends. Thus, the (i+1)st appointment in S' must be in A when the algorithm selects its (i+1)st appointment. The algorithm selects the appointment in A with the lowest end time, giving us f(i+1,S) $\leq$ f(i+1,S').Thus, our greedy algorithm chooses the subset of appointments that will help maximize the number of students we can help.
Time complexity is $O(nlogn)$ since the appointments are not sorted, and we go through the array to compare.

2. While your algorithm is clearly efficient and can proveably help the most students, you begin to receive complaints from students that you didn't help (i.e., their appointment was not part of the optimal solution). One of the students even offers to pay extra, which gives you a great idea. You will now allow students to make a donation to your favorite charity to make it more likely that their job will be selected. Let each appointment in this new set of appointments A be a triple $(start_i, end_i, donation_i)$ of start and end times and donation amounts where $end_i > start_i$ and $donationu_i > 0$. You now need to update your algorithm to handle these donations along with the requested appointment times. In this new environment, you are trying to maximize the amount of money you raise for your charity.

(a) (2 points) Give a specific case were your greedy algorithm would fail.

If there were two appointments;
Appointment 1: start = 1, end = 3, donation = 1
Appointment 2: start = 2, end = 4, donation = 100

The greedy algorithm would schedule the first appointment since it has the earliest end time, although the donation is greater for appointment 2.

(b) (5 points) Give a recursive algorithm that would solve this new case.
(help from geeksforgeeks.org)

Idea: Sort appointments in ascending order by end time. Represent each appointment by its donation. Choose subset of the values of the max sum of donations so none of the chosen appointments overlap. p(j) = largest i < j such that interval i doesn't overlap with j (its the furthest right interval that is compatible with j). Then,

a) If the first appointment is scheduled:
remove all appointments that overlap
recurs on remaining appointments.
b) if the first appointment isn't scheduled:
remove first job
recurs on remaining appointments.

Return max of a and b.

Pseudocode:

```
findMaxRec(A,n)                                  //assuming A is already sorted
    if(n==1) return A[n-1].donation              //base case
    S = empty
    profitInclude = A[n-1].donation              //find max when current included
    i = nonConflict(A,n)                         // to find p(j)
    if(i ! = -1)
       S.append(A[n])
       profitInclude += findMaxRec(A,i+1)
    profitExclude = findMaxRec(A, n-1) //find max when current not included
    totalMax = max(profitInclude, profitExclude)
return (S, totalMax)

nonConflict(A,i)
    for(j = i -1; j ≥ 0; j–)
       if(A[j].end ≤ A[i-1].start
          return j
    return -1
```

If nonConflict always returns a previous appointment it calls findMaxRec twice, making the time complexity $O(n * 2^n)$

(c) (3 points) Add memoization to this algorithm.

Pseudocode:

```
findMaxRec(A,n)
        if(n==1) return A[n-1].donation
        S[] = empty
        table = new int[n]
        table[0] = A[0].donation

        //memoize entries using recursion
        for(i = 1; i < n; i++)
           profitInclude = A[i].donation

        p = nonConflict(A,i)
        if(l ! = -1)
           profitInclude += table[l];
           S.append(A[i]

        table[i] = max(profitInclude, table[i-1])
result = table[n-1]
return(S, result)
```

(d) (10 points) Give a bottom-up dynamic programming algorithm.

Idea: Sort jobs ascending end times. For each i in A from 1 to n, find the max value of the appointments from the subsequence of appointments [0...i] by comparing the inclusion of appointment i to the schedule to the exclusion of appointment i to the schedule and taking the max of these. Use binary search to find the latest appointment which does not conflict

Pseudocode:

```
findMax(A , n)
        profit[] = new int[n+1]
        q[] = new int[n]
        Sort(A, A+n, apptComp)
        q[0] = -1          //find p(j)
       for (i = 1; i < n; i++)
           q[i] = binarySearchCompatible(A, i)

        profit[0] = 0
        for(j = 1; j ≤ n; j++)
           profitExclude = profit[j-1]
           profitInclude = A[j-1].donation
           if(q[j-1] != -1)
              profitInclude += profit[q[j-1]+1]
           profit[j] = max(profitInclude, profitExclude)
        return profit[n]

binarySearchCompatible(A,i)
        low = 0, high = i -1
        while( low ≤ high)
           mid = (low + high) / 2
           if(A[mid].end ≤ A[i].start)
              if(A[mid+1].end ≤ A[i].start)
                 low = mid + 1
              else
                 return mid
           else
              high = mid -1
        return -1
```

Since this algorithm uses binary search, the time complexity is O(n log n)

Name: Maura Kieft
ID: 103947905

3. (30 pts) The cashier's (greedy) algorithm for making change doesn't handle arbitrary denominations optimally. In this problem you'll develop a dynamic programming solution which does, but with a slight twist. Suppose we have at our disposal an arbitrary number of *cursed* coins of each denomination $d_1, d_2, \ldots, d_k$, with $d_1 > d_2 > \ldots > d_k$, and we need to provide n cents in change. We will always have $d_k = 1$, so that we are assured we can make change for any value of n. The curse on the coins is that in any one exchange between people, with the exception of $i = k-1$, if coins of denomination d_i are used, then coins of denomination d_{i+1} *cannot* be used. Our goal is to make change using the minimal number of these cursed coins (in a single exchange, i.e., the curse applies).

(a) (10 points) For $i \in \{1, \ldots, k\}$, $n \in \mathbb{N}$, and $b \in \{0, 1\}$, let $C(i, n, b)$ denote the number of cursed coins needed to make n cents in change using only the last i denominations $d_{k-i+1}, d_{k-i+2}, \ldots, d_k$, where d_{k-i+2} is allowed to be used if and only if $i \leq 2$ or $b = 0$. That is, b is a Boolean "flag" variable indicating whether we are excluding the next denomination d_{k-i+2} or not ($b = 1$ means exclude it).Write down a recurrence relation for C and prove it is correct. Be sure to include the base case.

I don't know.

(b) (10 points) Based on your recurrence relation, describe the order in which a dynamic programming table for $C(i, n, b)$ should be filled in.

I don't know.

(c) (10 points) Based on your description in part (b), write down pseudocode for a dynamic programming solution to this problem, and give a Θ bound on its running time (remember, this requires proving both an upper *and* a lower bound).

I don't know.